Maintenance of Embedded Systems: Supporting Program Comprehension Using Dynamic Analysis

Jonas Trümper Stefan Voigt Jürgen Döllner
Hasso-Plattner-Institute - University of Potsdam, Germany
{*jonas.truemper*|*stefan.voigt*|*juergen.doellner*}*@hpi.uni-potsdam.de*

***Abstract*—Maintenance of embedded software systems is faced with multiple challenges, including the exploration and analysis of the actual system's runtime behavior. As a fundamental technique, tracing can be used to capture data about runtime behavior as a whole, and represents one of the few methods to observe and record data about embedded systems within their production environments.**

In this paper we present a software-based, function-boundary tracing approach for embedded software systems. It uses static binary instrumentation, which implies only lightweight memory and performance overheads. To further reduce these overheads, instrumentation can be configured per trace, i.e., activated only for a specified group of functions without having to recompile the system. The technique can be characterized by its robust implementation and its versatile usage. It is complemented by a visualization framework that allows for analysis and exploration of a system's runtime behavior, e.g., to examine thread interaction. To show the technique's applicability, we conclude with a case study that has been applied to an industrial embedded software system.

I. Introduction

Software maintenance is known to be a major factor for costs in development of complex software systems [1], [2]. Therein, a large amount of effort is spent for *program comprehension* [3], [4] due to the size of a system's implementation and software aging [1]. With rapid growth and widespread use of embedded systems[1], there is also a growing demand for specialized techniques and tools supporting maintenance of these systems.

One reason for this demand is that *program comprehension of embedded systems* is faced with further problems compared to comprehension of general-purpose systems. Most important, means for observing the runtime behavior are severely limited. *Traditional debuggers* only yield snapshots of a system's state, i.e., they do not capture data about its runtime behavior as a whole and require a hardware interface to be present. Moreover, gathering this data typically requires halting a system's execution, thereby significantly altering its timing behavior [5], which is referred to as the *probe effect* [6]. Supplementary techniques such as, *simulators* or *special hardware* for obtaining data on a system's behavior, are often not applicable. Simulators, for instance, may fail due to missing simulation drivers for sensors or actors. Profilers and static checkers (e.g., Lint [7]) provide only aggregated data, which are of limited use for detailed analysis of runtime behavior. Manual workarounds, such as debug log output [8], are tedious and mostly prone to the probe effect. Another downside is that a complete cycle of building, linking and transferring the software to the target system is involved for reconfiguring log output.

Tracing, which allows for recording a system's runtime behavior, can be applied to overcome the described situation (1 and 2 in Fig. 1). However, analysis of resulting trace data is typically non-trivial due to their sheer size [9]. Visual analysis of trace data provides a means to cope with massive data and supports maintenance tasks such as program comprehension [9], debugging [10], and performance optimization [11] (3 and 4 in Fig. 1). Yet, for embedded systems, tracing facilities based on additional hardware, e.g., an *Embedded Trace Macro Cell (ETM Cell)* or an *In Circuit Emulator (ICE)*, are not provided for each processor and are typically very expensive. Software-based tracing techniques exist for general-purpose systems [12]–[15], but we identified a lack of versatile and cost-efficient solutions for embedded systems. Hence, a suitable tracing technique for these systems allows for bridging the gap, i.e., facilitate debugging and program comprehension of embedded systems using existing visual analysis techniques. Furthermore, the risk of encountering probe effects can be lowered if systems can be designed with the permanent injection of probes in mind [16]. Such instrumented software could also be delivered to customers if the resulting overhead of the instrumentation is low [16].

The main contributions of this paper are: (1) A concept for tracing and visualizing the runtime behavior of embedded systems. The proposed software-based tracing technique records function[2]-boundary traces (function entry, function exit *events*) and does not require a debug interface to be present. (2) We demonstrate the concept's feasibility by a prototype implementation that supports tracing of embedded single-threaded and multithreaded C/C++ software systems. (3) We show its usage and benefits by a conducted case

[1]In contrast to general-purpose systems, embedded systems are designed to accomplish a specific task.

[2]The term function is used interchangeably with the terms method, procedure, routine, etc.

study on an embedded software system (1.7 million lines of code) of an industrial partner.

II. Related Work

Software-based tracing techniques that are suitable for embedded systems use *static binary instrumentation* (before runtime) or *dynamic binary instrumentation* (at runtime). Existing representatives of the former, however, either suffer from limited applicability by using a custom pre-compiler [17], [18] to implement source-to-source transformations, drastically increase a binary's size [14] or rely on modifications to the operating system's source code [19] and typically require rebuilding the binary (or operating system) for any adjustments to the instrumentation.

Techniques using dynamic binary instrumentation embed code into the binary at runtime [20]. By contrast, dynamic binary instrumentation typically results in significant runtime overhead due to just-in-time recompilation. Substantial space overhead is added by code cache(s) for instrumented portions of the original code. Alternatively, callbacks of the operating system [16] can be used to monitor interrupts and scheduling events. This technique, though, faces strict limitations with respect to measuring performance of specific functionality or capturing function-boundary traces.

Orthogonal approaches aim at computing an instrumentation scheme that ensures low runtime overhead [21], [22]. These mechanisms could be used to simplify selective instrumentation with our instrumentation concept. Similarly, sampling can be used [18] to keep runtime overhead low. That, however, looses precision and may miss executions of functions of specific interest in debugging scenarios. Other approaches aim to bring reproducibility to cyclic debugging using interrupt checksums [23], but still require cyclic debugging.

Hybrid hardware-/software-based and purely hardware-based techniques differ fundamentally from our technique by requiring hardware support. Hybrid solutions can use re-programmable microcodes of processors [24] to trigger tracing functionality whenever such a re-programmed microcode is executed. An industrial hybrid tracing-technique is distributed by Rapita Systems[3].

Pure hardware-based solutions for performance optimization include instruction counters [25] or signal analysis techniques [26]. In addition, record/replay techniques allow for reproducible cyclic debugging [27]–[29]. Yet, besides overcoming the problem of non-deterministic behavior, these approaches have the same shortcomings as traditional debuggers. Techniques that are able to record function-boundary traces similar to our approach make use of new processor generations containing an embedded ICE [30] or the ARM CoreSight [31] as an enhanced ARM processor architecture that comprises an embedded macro cell specifically designed for tracing purposes.

[3]http://rapitasystems.com/products/RTBx/, last accessed 04/02/2011

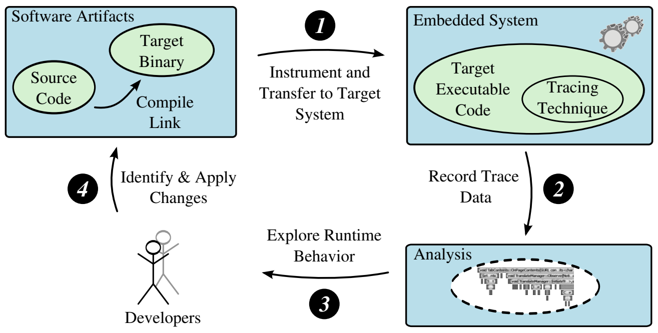


Figure 1. Example of how tracing can facilitate software-maintenance tasks: (1) Instrumentation, (2) trace recording at runtime, (3) visual analysis of trace data – developers gaining insights, (4) applying necessary changes to software artifacts.

III. Embedded Systems: Requirements for Software-based Tracing Concepts

Application domains of embedded systems typically exhibit fundamental differences to those of general-purpose systems that directly influence feasibility of tracing concepts. We identified the following requirements to be considered by software-based tracing concepts for embedded systems.

R1) Cope with Strictly Limited Resources: Embedded systems typically run on strictly limited resources compared to general-purpose systems, which includes the amount of available memory. Consequently, a tracing technique for embedded systems should try to minimize the memory overhead. This includes the tracing logic's binary size, its working memory and memory needed to store runtime events. Obviously, it should lead to minimal processing overhead as well.

R2) Run in the Same Process: In the context of embedded systems, process management may not be supported by the hardware or the operating system. To guarantee a high applicability of a tracing concept, it should be able to run in the same process as the instrumented software.

R3) Cope with Limited System Connectivity: Although most currently available processors for embedded systems feature a debug interface, it cannot be assumed that (a) the debug interface has sufficient bandwidth for real-time interaction with a probe, or (b) that the target system is connected via another interface with sufficient bandwidth. Hence, trace data likely cannot be retrieved from the embedded system at runtime. Further, to be applicable in production environments, the tracing concept should be able to obtain information on a system's behavior without an external interface being permanently present.

R4) Be Independent of a Specific Operating System Family: There is a wide variety of operating system families for embedded systems. To guarantee wide applicability, the concept should not depend on the availability of a specific operating system family.

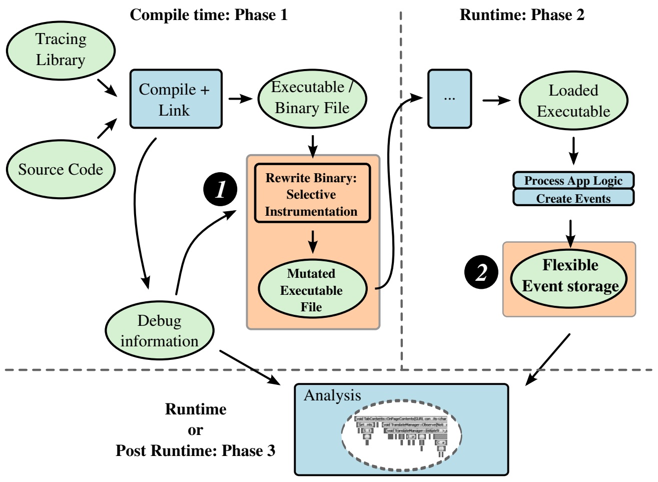


Figure 2. Software-based tracing concept for embedded systems.

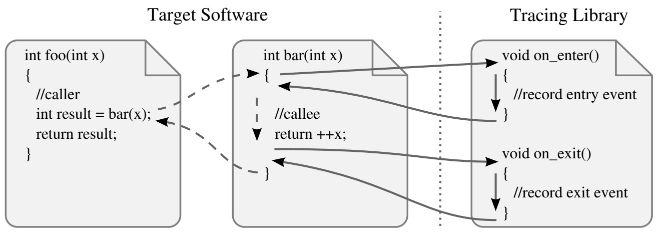


Figure 3. Control flow within the instrumented target system: Original control flow (dashed arrows), control flow redirected by hooks (solid arrows).

IV. Tracing Concept for Embedded Systems

Our concept combines a number of techniques to fulfill the mentioned requirements. The concept addresses three phases of the analyzed system: Compilation, execution and post-runtime (Fig. 2).

Phase 1 - Instrumentation: Before the binary is executed, a library that comprises tracing functionality is added. The binary is modified to trigger the tracing library at specified points in execution, e.g., during function entry or exit. For that, a compiler functionality to redirect the given control flow at each function entry [32] is exploited. If a specific compiler switch is given, function bodies are compiled with modified prologs. Therein, control flow is redirected by a *hook* to a *hook implementation* that records trace data (Fig. 3). This redirection functionality is supported by most modern compilers, so our concept is at least applicable to hardware platforms supported by these compilers.

The proposed tracing technique still records both function entry *and* exit events by dynamically instrumenting function exits, so that performance and timing analysis becomes possible. The advantage of dynamically instrumenting exit hooks is that it (a) enables tracing of function exit events on compilers that exclusively support insertion of function entry hooks, and (b) simplifies selective instrumentation, because only entry hooks need to be disabled – replaced by NOP instructions – for deactivating a function's instrumentation.

In addition, developers can insert custom hooks into a system's source code to record *user-defined events* with payload data, e.g., to mark entry/exit of critical sections in recorded trace data. In contrast to common log output, these events become an integral part of trace data, allowing for their visual analysis and exploration.

To reduce runtime overhead and trace size, tracing techniques for general-purpose systems may perform analysis of the recorded trace data at runtime to automatically disable tracing of specific functions. For instance, frequently executed utility functions, e.g., string concatenation, typically contribute to a large amount to a trace's size, but not to a better comprehensibility of the data [33]. Such approaches permit to obtain lower performance overhead even with costly tracing implementations: The performance overhead is only added to costly, less frequently executed, functions and thus results in a lower *relative* and therefore lower *absolute* overhead. Similarly, trace compression [34] is a means to reduce the size of recorded data.

Selective instrumentation at runtime, however, requires that a system's code can be modified externally using a debugger interface or that a system has sufficient computational power available to perform the runtime analysis itself. The latter requirement holds as well for trace compression. Even more so, it is important to keep the amount of recorded data low: (a) The size of the trace-data storage may be very limited, (b) recording less data also means lower runtime overhead, and (c) the tracing technique should not distort analysis results and should be suitable for every-day use, even for normal builds.

Thus, we shift selective instrumentation to the first phase (1 in Fig. 2) by modifying the binary prior to its execution. Although compilers typically provide means to selectively instrument on a per-file basis, we have experienced that this is too coarse-grained in practice: Applying selective instrumentation per function is necessary to achieve sufficient control over recorded trace data and, at the same time, runtime overhead. Binary patching is used to disable instrumentation of all functions after compilation, resulting in a modified copy of a linked executable file that is then loaded and executed. This binary can be used for normal runs in production environments, implying almost no performance overhead. Instrumentation for a selected group of functions can later be enabled again, allowing for reconfiguring the instrumentation without recompiling the binary.

An input set of functions for selective instrumentation can typically be gathered from developer knowledge. If not, however, a similar approach to the aforementioned utility function classification is possible: Based on measured execution frequencies, functions are iteratively de-selected from instrumentation until sufficiently low overhead is achieved. Other alternatives are existing approaches that compute an

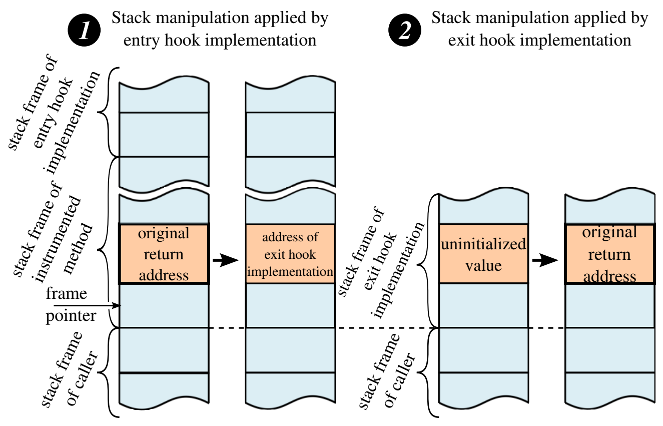

1
Stack manipulation applied by entry hook implementation
2
Stack manipulation applied by exit hook implementation
stack frame of entry hook implementation
stack frame of instrumented method
original return address
address of exit hook implementation
frame pointer
stack frame of caller
stack frame of exit hook implementation
uninitialized value
original return address
stack frame of caller

Figure 4. Return address manipulation at runtime. (1) Entry hook implementation manipulates a stack frame of an instrumented function to redirect the control flow to the exit hook implementation. (2) Exit hook implementation manipulates its own stack frame to proceed with normal execution.

optimal instrumentation [21], [22] prior to execution.

Phase 2 - Tracing: After instrumentation, the binary and the tracing library are loaded into the target system and executed. Instrumentation of function exits is performed at runtime using return address manipulation. The *entry hook implementation* manipulates the *stack* of the respective thread at runtime so that an instrumented function does not return to its original caller, but instead returns to a second instrumentation function, the *exit hook implementation* (1 in Fig. 4). After the exit hook implementation finishes, it returns the control flow to the original caller (2 in Fig. 4). Both hook implementations use a shared data structure, the *shadow stack*, for returning to the original control flow and to recognize exceptions properly.

The application domain and execution environment of an embedded system define available storage for trace data. Thus, the concept must not rely on virtually unlimited storage size and support relocation of storage depending on the concrete embedded system to instrument (2 in Fig. 2).

Phase 3 - Analysis: Recorded trace data is analyzed *offline*, i.e., after the system ran. Access to that data is possible by transferring it from an embedded system to a general-purpose system. Any means that support transferring arbitrary data are suitable, such as memory card, wired/wireless network or a debug interface. Debug information is used to reconstruct human-readable names from raw trace data.

V. Case Study

We discuss a case study that we conducted on an industrial embedded system: An electronic postage system [35]. The multithreaded software system for its control unit consists of approximately 1.7 million lines of code, with the majority in C++ and some portions in C, and Assembler. It runs on an ARM-based circuit-board and is compiled using the Green Hills Software's MULTI v4.2.4, which produces an ELF binary. Express Logic's ThreadX 5 serves as the operating system. While development took about 2 years, maintenance of the code base is ongoing for 3 years.

As part of a maintenance task involving performance optimization, developers had two technical questions regarding comprehension of the system's runtime behavior:

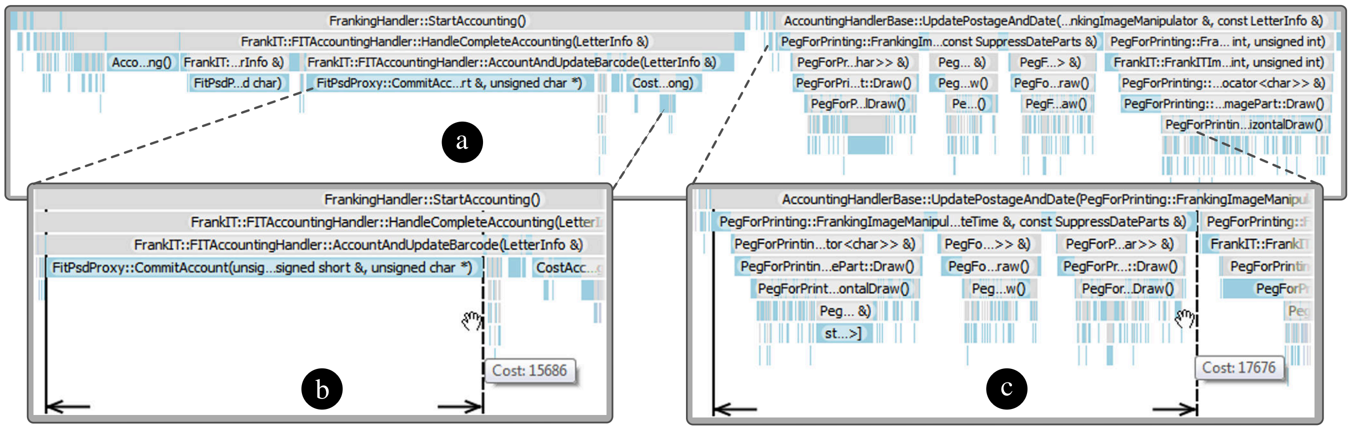
1) Is there any time-critical execution path of which execution time in the current implementation is close to violating its permitted execution time?
2) Which portions of the concerned code have significant potential for further performance optimization?

Using log output to answer the first question turned out to be troublesome. Running the system with a few added log statements already caused a violation of a timing constraint. Numerous time-intensive build-and-run cycles would have been necessary to obtain a set of log statements that would not violate any timing constraints and provide a reliable answer. We thus applied the tracing technique by adding custom events for marking time critical sections to: (a) The start of a time critical action, (b) the end of this action and (c) the point in execution where the action needs to be finished. Alternatively, it would also be possible to instrument methods close to these points in execution to gather estimate measurements without the need for recompilation, but we decided to use custom events for simplicity.

With the tracing technique being lightweight, a trace was recorded successfully. We identified the construction of the *print image*, which is used for franking a letter, as a critical path. The construction finished just in time before the image was needed for printing it to a moving envelope that passes the print head.

We asked expert developers of our industrial partner which findings they expected for the second question with respect to construction of the print image. They were aware that synchronous communication with a security device would consume a considerable share of the overall execution time. However, since this procedure was already optimized from two communication cycles down to a single cycle, only minor performance improvement – at the expense of maintainability – is possible. As it was further unclear how much time was consumed by the remaining code, we used the visualization framework to answer that question and to verify the developers' expectations. Functions were selectively instrumented if developers considered them to be relevant for the question and if they were not categorized as utility functions (Section IV).

The visual representation [36] of recorded trace data allowed for identifying how functions are composed to implement high-level tasks (a in Fig. 5). Based on this, with the developers we were able to assess whether execution time of a respective task was reasonable. We found that the developers' expectations regarding communication with the security device was correct (b in Fig. 5). Complementarily, we discovered that a large share of the remaining time for



FrankingHandler::StartAccounting()
FrankIT::FITAccountingHandler::HandleCompleteAccounting(LetterInfo &)
Acco...ng()
FrankIT:...rInfo &)
FrankIT::FITAccountingHandler::AccountAndUpdateBarcode(LetterInfo &)
FitPsdP...d char)
FitPsdProxy::CommitAcc...rt &, unsigned char *)
Cost...ong)
AccountingHandlerBase::UpdatePostageAndDate(...nkingImageManipulator &, const LetterInfo &)
PegForPrinting::FrankingIm...const SuppressDateParts &)
PegForPrinting::Fra... int, unsigned int)
FrankIT::FrankITIm...int, unsigned int)
PegForPrinting::...ocator<char>> &)
PegForPrinting::...magePart::Draw()
PegForPrintin...izontalDraw()
a
FitPsdProxy::CommitAccount(unsig...signed short &, unsigned char *)
CostAcc...
Cost: 15686
b
AccountingHandlerBase::UpdatePostageAndDate(PegForPrinting::FrankingImageManipu
PegForPrinting::FrankingImageManipul...teTime &, const SuppressDateParts &)
PegForPrintin...tor<char>> &)
PegForPrintin...ePart::Draw()
PegForPrint...ontalDraw()
Cost: 17676
c

Figure 5. a) Visualization of a thread's trace data captured during franking a letter. Stack grows from top to bottom, time progresses from left to right. Left part, phase 1: Calculating postage data and communication with the security device. Right part, phase 2: Generating print image from postage data. b) Detailed view on phase 1: Measuring execution cost (in μs) for communicating with the security device. c) Detailed view on phase 2: Measuring execution cost (in μs) for updating date and time of the print image.

generating a print image is spent on updating the portion that holds the current date (c in Fig. 5). A conclusion from these findings is that the date element of the print image is currently processed for each letter. This element can, however, be pre-computed and cached. By that, a significant reduction in execution time for each franked letter was achieved.

VI. Memory and Performance Overhead

We first show the lightweight memory and performance overheads of our approach and further evaluate the compile-time effort for re-configuring instrumentation.

A. Memory Overhead

Memory overhead is added by (1) stack frames for functions of the tracing library, (2) per-thread shadow stacks, and (3) the buffer for trace data. The binary representation of the tracing library itself and required hooks add further memory overheads. The latter two, however, are typically insignificant compared to the size of the instrumented system and will not be discussed in this paper.

Tracing techniques with hook implementations that are active while an instrumented function is executed, add memory overhead per stack frame. In contrast, our technique adds only a constant overhead for each thread's stack, since our hook implementations are only active before and after an instrumented function is executed.

Further, each per-thread shadow stack has a predefined, fixed size. The number of instrumented functions on the program stack may exceed this size, but the exit hook implementation will not be executed in this case. Each shadow stack element uses 12 bytes to store caller, callee, and respective stack pointer, 4 bytes each.

A compact 21-byte data structure encodes each event to keep space and runtime overheads low; lookup of function names using debug symbols is done offline: In the trace data, functions are referred to by their memory address. Besides minimizing the size of recorded data, the speed of recording is crucial. As others have shown [16], storing trace data inside a circular RAM buffer is fast enough for most embedded systems. We therefore chose the same storage for the trace data. In the current implementation, the instrumented system holds an extra 512 Kbytes for the RAM buffer that stores the most recent 24,966 events $\left(\frac{512\times1024\ bytes}{21\ bytes} = 24,966\right)$.

B. Performance Overhead and Selective Instrumentation

We measure the performance overhead of our tracing library used in the case study (Section V) by comparing the execution time (Fig. 6) of the following configurations: (a) Original, (b) instrumented, but tracing disabled using NOPs, (c) selectively instrumented 22% of all functions (~24,500). Measurements using a fully instrumented binary failed because timing constraints were violated, causing security mechanisms of the system to halt execution. For each configuration, execution time was measured 10 times.

Configuration c) shows an overhead of 18%, which is a non-negligible slowdown – especially if a larger portion of a system would be instrumented and executed. Yet, when analyzing the recorded trace for execution frequencies of functions (Fig. 7), a high peak for function 10 indicates that performance overhead could be further reduced in exchange for little loss of detail by disabling this single function. In addition, during development of the prototype implementation, our focus was mainly on feasibility. Thus, we see potential for further performance improvements using inlining, less copy operations etc. More importantly, execution with deactivated tracing causes an overhead of merely 2%. Hence, such binaries can typically be used as a replacement for the original binaries.

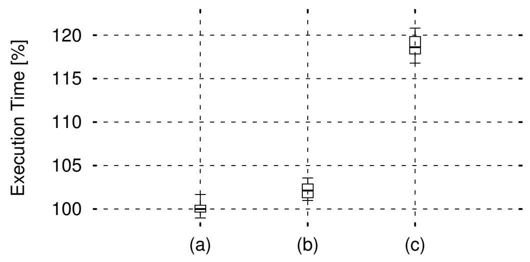


Figure 6. Performance overhead comparison: Normal run of the system with no instrumentation (a), instrumented, but disabled (b), and selectively instrumented (c). The 100% mark is the mean execution time of the original binary.

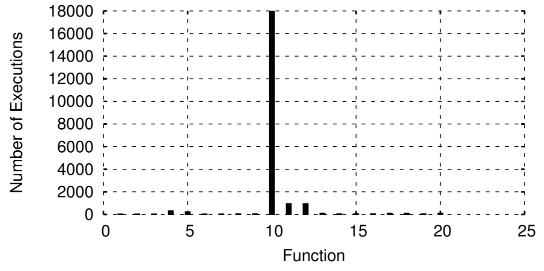


Figure 7. Execution frequencies: A high peak for function 10 indicates that performance overhead could be reduced even further in exchange for some loss of detail in recorded trace data.

The performance of selective instrumentation itself was measured as the time required to replace all hook calls by NOP instructions: On average, it took about 3.4 seconds with a standard deviation of $\sigma = 0.058$ on a 2 GHz CPU with 2 GB RAM. Re-activating the hooks by replacing the NOP instructions again is about the same complexity. Hence, selective instrumentation could be integrated as an additional, regular build step.

VII. Threats to Validity and Limitations

The conducted case study is prone to some threats. First, available computational power and available writable storage may restrict the maximum number of hooks that can be inserted or even prevent instrumentation at all. If no additional processing power is available for executing tracing functionality – without violating timing constraints – then observing a system's behavior will fail at some point. In that case, however, it is still possible to analyze data that was captured up to this point. Second, the case study demonstrated the technique's usefulness with regard to timing analysis. Investigation of further possible maintenance tasks is still to be conducted. Third, the case study is limited to capturing runtime data of a single process.

We identified the following restrictions of the technique: (a) The concept relies on storage for per-thread shadow stacks. If that is not available, no function exit events can be instrumented. (b) Selective instrumentation without recompiling and relinking is only supported for compiler-generated hooks, so reconfiguring custom events involves rebuilding (parts of) a binary. (c) Recorded data is influenced by the tracing technique itself as the technique is executed within a target system. That is, if frequently executed functions are instrumented, relative tracing overhead distorts measured data to some degree.

VIII. Conclusions

Embedded systems pose numerous challenges to program comprehension, including non-determinism, complex debugging setups, and shortcomings in simulation of sensors or actors. A lack of appropriate tools renders overcoming these challenges a time-consuming and error-prone task in software maintenance. Existing tracing techniques that aim to provide better tool support, however, rely on dedicated interfaces, are not available for all processor types, and are hardly applicable in production environments.

We have presented a software-based technique for tracing runtime behavior of embedded software systems that meets the specific requirements of embedded systems. We further conducted a case study on an industrial embedded system that demonstrates how the technique works within a practical maintenance situation. The technique enables developers to replace common debugging workarounds, such as log output and guesswork, by an efficient trace analysis technique. Independence of a debug interface is a key characteristic of the technique and enables its usage in production environments as well. As a result, such integration has potential to ease another common problem of software maintenance: Reproducing bugs that are encountered in a customer's production environment.

As future work, we plan to enable longer trace runs by triggering serialization of trace data to persistent storage while a target system is idle. In addition, we want to integrate approaches that help automate selective instrumentation for given storage size and software system. Complementary, enriching trace data by operating system events, e.g., scheduling, would provide valuable extra information.

Acknowledgements

We thank Francotyp-Postalia GmbH (http://www.francotyp.com) for their cooperation and provision of embedded systems, and, in particular, Dirk Rosenau for fruitful discussions on implementation details. We further thank Software Diagnostics GmbH (http://www.softwarediagnostics.com) for providing their visualization framework. This work was supported by the ZIM program of the Federal Goverment of Germany (BMWi).

References

[1] D. L. Parnas, "Software aging," in *Proc. ICSE*, 1994, pp. 279–287.

[2] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Prof.*, vol. 2, no. 3, pp. 17–23, 2000.

[3] V. R. Basili, "Evolving and packaging reading technologies," *JSS*, vol. 38, no. 1, pp. 3–12, 1997.

[4] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code," in *Proc. ICSM*, 1992, pp. 200–205.

[5] M. Lindahl, "Analyzing real-time systems with hardware trace," *C++ Users Journal*, Mar. 2005.

[6] J. Gait, "A probe effect in concurrent programs," *Software Pract Exper*, vol. 16, pp. 225–233, Mar. 1986.

[7] S. Johnson, "Lint, a c program checker," Bell Laboratories, Tech. Rep. 65, Dec. 1977.

[8] P. Horwood, S. Wygodny, and M. Zardecki, "Debugging multithreaded applications," *Dr. Dobb's Journal of Software Tools*, vol. 25, no. 3, pp. 32, 34–37, Mar. 2000.

[9] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," *IEEE JSE*, vol. 81, no. 12, pp. 2252–2268, 2008.

[10] K. Mehner, "Trace-based debugging and visualisation of concurrent java programs with uml," Ph.D. dissertation, Universität Paderborn, 2005.

[11] S. S. Shende and A. D. Malony, "The tau parallel performance system," *SAGE IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.

[12] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proc. PLDI*, 2007, pp. 89–100.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005, pp. 190–200.

[14] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Proc. ISPASS*, 2010, pp. 175–183.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proc. ECOOP*, 2001, pp. 327–353.

[16] J. Kraft, A. Wall, and H. Kienle, "Trace recording for embedded systems: Lessons learned from five industrial projects," in *Runtime Verification*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6418, pp. 315–329.

[17] O. Spinczyk, D. Lohmann, and M. Urban, "Aspectc++: an aop extension for c++," *Software Developer's Journal*, pp. 68–76, May 2005.

[18] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California, Berkeley, Dec. 2004.

[19] A. Schmidt and M. Schöbel, "Analyzing system behavior: How the operating system can help," in *Proceedings of GI Jahrestagung (2)*, 2007, pp. 261–267.

[20] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proc. CASES*, 2006, pp. 261–270.

[21] S. Fischmeister and P. Lam, "Time-aware instrumentation of embedded software," *IEEE TII*, vol. 6, no. 4, pp. 652–663, Nov. 2010.

[22] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," in *Proc. ICSE*. ACM, 1999, pp. 277–284.

[23] D. Sundmark and H. Thane, "Pinpointing interrupts in embedded real-time systems using context checksums," in *Proc. ETFA*, Sep. 2008, pp. 774–781.

[24] D. H. Barnes and L. L. Wear, "Instruction tracing via microprogramming," in *Proc. MICRO*, 1974, pp. 25–27.

[25] T. A. Cargill and B. N. Locanthi, "Cheap hardware support for software debugging and profiling," in *Proc. ASPLOS*, ser. ASPLOS-II, 1987, pp. 82–83.

[26] A. Mink and G. Nacht, "Performance measurement of a shared-memory multiprocessor using hardware instrumentation," in *Proc. HICSS*, vol. 1, Jan. 1989, pp. 267–276.

[27] B. Plattner, "Real-time execution monitoring," *IEEE TSE*, vol. SE-10, no. 6, pp. 756–764, 1984.

[28] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *IEEE TSE*, vol. 16, pp. 897–916, Aug. 1990.

[29] L. Moore and A. Moya, "Non-intrusive debug technique for embedded programming," in *Proc. ISSRE*, Nov. 2003, pp. 375–380.

[30] Y.-L. Jeang, T.-s. Wey, H.-Y. Wang, and C.-T. Chen, "A new real-time address tracer for embedded microprocessors based on preprocessing," in *Proc. ICICIC*, 2007, pp. 263–266.

[31] W. Omre, "Debug and trace for multicore socs," ARM Limited, Tech. Rep., 2008.

[32] S. Voigt, J. Bohnet, and J. Döllner, "Object aware execution trace exploration," in *Proc. ICSM*, Sep. 2009, pp. 201–210.

[33] A. Hamou-Lhadj, "Techniques to simplify the analysis of execution traces for program comprehension," Ph.D. dissertation, University of Ottawa, 2005.

[34] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proc. ICSE*, 2001, pp. 221–230.

[35] G. Bleumer, *Electronic Postage Systems: Technology, Security, Economics*, ser. Advances in Information Security, S. Jajoda, Ed. Springer Berlin, 2006.

[36] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *Proc. SOFTVIS*, 2010, pp. 133–142.